Python propose un grand nombre de fonctions utilisables par les programmeurs.

1 Fonctions accessibles directement

Certaines fonctions sont accessibles directement dans l'interpréteur (*built-in functions*), comme la fonction `type` vue au chapitre précédent.

Par exemple, la fonction `len()` renvoie le nombre de caractères d'une chaîne de caractères.

```
>>> len('informatique')
12
>>> len('')
0
```

La fonction prédéfinie `min()` renvoie le plus petit de ses paramètres, le nombre de paramètres pouvant varier. On peut donc écrire :

```
>>> min(12, 6)
6
>>> min(3, 4, 1, 2, 4)
1
```

On peut imbriquer les appels de fonctions :

```
>>> min(len('informatique'), len('Python'))
6
```

Certaines fonctions sont sans paramètre. On écrit alors simplement le nom de fonction suivi de parenthèses.

Par exemple, la fonction `dir()` renvoie une liste des variables (pré-)définies :

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'd', '
```

2 Modules Python

D'autres fonctions sont proposées via des **bibliothèques**, ou **modules**. Elles ne sont pas accessibles directement et doivent être **importées**.

Python fournit de très nombreux modules. Nous en découvrirons quelques-uns au fur et à mesure de ce semestre.

2.1 Quelques modules

Voyons ci-dessous un bref aperçu de deux modules que nous utiliserons dans nos exemples.

2.1.1 Le module `math`

Le module `math` de Python contient les définitions de nombreuses fonctions mathématiques telles que *sinus* – `sin()` –, *cosinus* – `cos()` -, *tangente* – `tan()` –, *racine carrée* – `sqrt()` –, etc.
Il contient également des constantes telles que π, noté `pi`.

2.1.2 Le module `random`

Le module `random` de Python regroupe les fonctions de hasard (tirage au sort, génération aléatoire). Nous utiliserons les fonctions `choice` et `randint`.

`choice()` accepte par exemple une chaîne de caractères et renvoie un des caractères de la chaîne, choisi au hasard. On notera que la doc indique la précondition de la fonction, dans la phrase qui documente la fonction. La chaîne doit être non vide (de longueur non nulle).


```
>>> help(choice)
Help on method choice in module random:

choice(seq) method of random.Random instance
    Choose a random element from a non-empty sequence.
```

`randint()` accepte deux paramètres de type entier - bornes inférieure et supérieure d'un intervalle - et renvoie un entier, choisi au hasard dans cet intervalle, bornes comprises. Cette fois la précondition est moins claire :

```
>>> help(randint)
Help on method randint in module random:

randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.
```

Ces fonctions calculent un résultat de manière aléatoire donc d'un appel à l'autre le même appel renverra possiblement une valeur différente.

2.2 Importer les fonctions d'un module

L'utilisation d'une fonction d'un module nécessite une importation préalable.

Il est possible :

- d'importer un module
- d'importer une fonction particulière d'un module
- ~~d'importer toutes les fonctions d'un module~~ (mauvaise pratique)

L'importation n'est réalisée qu'une fois, habituellement en début de programme ou de session. De multiples appels à la fonction sont ensuite possibles.

2.2.1 Importer un module

Pour importer un module, par exemple le module `random`, on utilise la syntaxe suivante

```
import random
```

On identifiera les fonctions importées en préfixant leur nom par celui du module. Par exemple

```
>>> random.choice('aeiouy')
'u'
>>> random.choice('aeiouy')
'y'
```

ou

```
>>> random.randint(12, 42)
33
>>> random.randint(5, 5)
5
>>> random.randint(5, 3)
[...]
ValueError: empty range for randrange() (5, 4, -1)
```

On commence à se douter que la précondition de cette fonction est : la valeur de la borne inférieure de l'intervalle est inférieure ou égale à la valeur de la borne supérieure, ce qui est logique.

2.2.2 Importer une fonction particulière d'un module

Pour importer une fonction particulière d'un module, par exemple la fonction `choice()` du module `random`, on utilise la syntaxe suivante

```
from random import choice
```

On identifiera simplement la fonction importée par son nom. Par exemple

```
>>> choice('aeiouy')
'y'
```

Remarquons que les autres fonctions du module ne sont pas disponibles :

```
>>> randint(12, 42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'randint' is not defined
```

Le message d'erreur indique que le nom `randint` n'est pas défini.

Quelle solution préférer entre importer un module ou une fonction particulière ? C'est affaire de goût.

2.2.3 Importer toutes les fonctions d'un module

On peut vouloir importer l'ensemble des fonctions d'un module. On utilise alors la syntaxe :

~~`from random import *`~~

On identifiera alors simplement les fonctions importées par leur nom. Par exemple

```
>>> choice('aeiouy')
'o'
```

Cette manière de faire est une mauvaise pratique. Même si ça semble plus rapide au premier abord d'écrire `import *`, on s'en mordra les doigts par la suite.

Ce moyen d'importation « pollue » l'espace de noms : de nombreuses fonctions deviennent disponibles alors que nous n'allons pas les utiliser. Pour le constater, essayer `from math import *` dans la console de Thonny-l1test avec la fenêtre `Variables` ouverte !

De plus, quand on lit un module étranger pour tenter de le comprendre et qu'on tombe sur l'appel d'une fonction `foo` non définie dans le module, un `from truc import foo` est très aidant pour savoir où trouver la définition de `foo`.

Enfin, si on utilise `from A import *` et `from B import *` et que `A` et `B` contiennent chacun une fonction de nom `foo`, il y a conflit !

3 Documentation d'une fonction (rappel)

La documentation d'une fonction est un texte écrit en langue naturelle : les programmeur · es y décrivent brièvement la **spécification** de la fonction, c'est-à-dire une description plus ou moins formelle du "quoi" (ce que fait le fonction) et non du "comment".

La documentation est indispensable car elle indique aux utilisateurs et utilisatrices de la fonction son usage et son comportement sans qu'ils aient besoin de lire son code.

On trouve les documentations de Python sur `python.org` mais on peut aussi les afficher directement depuis la console grâce à la fonction `help()`. Pour obtenir l'aide sur la fonction prédéfinie `abs()`, on écrira

```
>>> help(abs)
Help on built-in function abs in module builtins:

abs(x, /)
    Return the absolute value of the argument.
```

Cette aide est interactive si elle tient sur plusieurs pages :

- on passe à la page suivante avec la touche « *espace* »
- on quitte l'aide en utilisant la touche « `q` »

On découvre :

- la notion de variable en informatique
- l'instruction d'affectation

1 Variables et affectation

Une variable est un nom qui va permettre de désigner une valeur.

Ce nom sera remplacé par la valeur associée à la variable.

L'opération d'**affectation** permet de lier une valeur à la variable.

La syntaxe de l'affectation (c'est-à-dire la manière d'écrire une affectation) est illustrée par l'exemple suivant :

```
>>> x = 2
```

On distingue trois éléments :

- une valeur, ici l'entier `2`
- le nom de la variable, ici `x`
- un signe `=` qui sépare le nom de la variable - en partie gauche - de la valeur - en partie droite -

Plus généralement la syntaxe de l'affectation est la suivante :

```
<nom_var> = <expr>
```

(en se rappelant que `<nom_var>` est à remplacer par un nom de variable).

NB La syntaxe de l'affectation peut perturber car elle ressemble à une égalité, alors qu'il n'est pas du tout question d'égalité. D'autres langages de programmation ont choisi `:=` comme symbole pour l'affectation. En Python ce symbole produira une erreur de syntaxe.

La **sémantique** de l'affectation indique quel effet a une affectation, quand on l'exécute. L'**exécution** de l'affectation se déroule en deux étapes :

1. la valeur de la partie droite de l'affectation est calculée
2. cette valeur devient la valeur associée à la variable

Suite à l'exécution de l'affectation précédente, nous pouvons observer cette association :

```
>>> x
2
```

L'association entre variable et valeur apparaît dans la mémoire associée (dans notre cas) au processus Python, initialement vide. Cette mémoire peut-être représentée par un tableau. Suite à l'affectation précédente, la mémoire contient :

x	2

La partie droite de l'affectation peut-être une expression.

```
>>> y = 3+4
```

En utilisant le même mécanisme, la valeur de la partie droite (l'expression `3+4`) est évaluée à la valeur `7`, puis la valeur `7` devient la valeur associée à la variable `y`.

```
>>> y
7
```

Suite à l'exécution de l'affectation précédente, la mémoire contient 2 associations:

x	2
y	7

Il est possible de procéder à une nouvelle affectation d'une variable déjà définie. L'association avec l'ancienne valeur est alors perdue. Par exemple :

```
>>> y
7
>>> y = 12
>>> y
12
```

Suite à l'exécution de l'affectation précédente, la mémoire contient toujours 2 associations:

x	2
y	12

L'expression en partie droite peut contenir des variables. Dans ce cas, l'évaluation de l'expression utilise la valeur associée à la variable dans la mémoire :

```
>>> x
2
>>> y
12
>>> z = x + y
>>> z
14
```

La mémoire contient alors :

x	2
y	12
z	14

L'expression en partie droite peut même utiliser l'ancienne valeur de la variable qui sera affectée :

```
>>> x
2
>>> x = x + 1
>>> x
3
```

L'exécution de l'affectation se déroule alors ainsi :

- évaluation de la partie droite, dans laquelle `x` est évaluée à `2`, qui produit la valeur `3`
- association de la valeur `3` à `x`

La mémoire contient alors :

x	3
y	12
z	14

NB Le signe `=` utilisé pour l'affectation n'a rien à voir avec le $=$ mathématique ! Que pourrait signifier l'équation $x = x + 3$?

NB On notera que dans l'exemple précédent l'affectation `x = x + 1` n'a pas modifié la valeur en mémoire de `z`. Il ne faut pas lire `z = x + y` comme "`z` est défini par `x + y`" (ce qui impliquerait que, quand la valeur associée à `x` ou `y` change, celle de `z` est recalculée), mais bien comme une association variable-valeur qui prend en compte les valeurs de `x` et `y` au moment où l'affectation est exécutée.

La partie gauche d'une affectation est syntaxiquement une variable :

```
>>> x = 3
>>> x + 1 = x
  File "<stdin>", line 1
    x + 1 = x
    ^^^^^
SyntaxError: cannot assign to expression here. Maybe you meant '==' instead of '='?
```

On ne sait pas pour le moment ce que signifie `==` (on le verra dans le prochain chapitre) mais on a bien compris qu'il y a une erreur de syntaxe.

La même erreur se produit si on confond l'identifiant de variable et la chaîne de caractères qui contient cet identifiant! Cette erreur est classique pour les débutant · es.

```
>>> "x" = 1
[...]
    "x" = 1
    ^^^
SyntaxError: cannot assign to literal here. Maybe you meant '==' instead of '='?
```

Si l'évaluation de l'expression en partie droite déclenche une erreur, la mémoire n'est pas modifiée :

```
>>> x
3
>>> x = x / 0
[...]
ZeroDivisionError: division by zero
>>> x
3
```

Que se passe-t-il quand on cherche à évaluer la valeur d'une variable à laquelle aucune valeur n'est associée en mémoire ?

```
>>> x = m
[...]
NameError: name 'm' is not defined
```

On découvre un nouveau nom d'erreur ! Comme précédemment, la valeur associée à `x` n'a pas été modifiée. Une telle erreur se produit aussi quand on tente de calculer la nouvelle valeur d'une variable en fonction de sa valeur précédente qui n'existe pas :

```
>>> e = e + 1
[...]
NameError: name 'e' is not defined
```

2 Instructions

Une affectation est une **instruction**.

Une instruction exprime un ordre, elle s'**exécute**. Son exécution modifie l'état de la mémoire. Elle n'a pas pour vocation d'être évaluée à une valeur.

Remarquons qu'une expression n'est pas une instruction. Elle ne modifie pas l'état de la mémoire. Par contre elle s'évalue à une valeur.

La ligne qui indique la valeur renvoyée par une fonction (`return ...`) est une instruction, qui ne peut être utilisée que dans le corps d'une fonction. Nous verrons d'autres instructions Python bientôt.

3 Séquence d'instructions, bloc de code

Nous avons vu la notion de bloc de code dans le chapitre précédent, dans le cas du corps d'une fonction.

D'une manière générale, on peut écrire un bloc de code qui contient une **séquence** d'instructions, c'est à dire une suite d'instructions qui se suivent. Pour le moment nous ne savons écrire que des séquences d'affectations:

Université de Lille
Variables et affectation
2025-2026
UE PROG, L1S1 MI
Département Informatique de la Faculté des Sciences et Technologies

```
x = 1     #1
y = x + 2 #2
```

La sémantique d'une séquence d'instruction est la suivante :

- en début de séquence la mémoire est vide
- on exécute successivement les instructions du haut vers le bas en modifiant la mémoire au fur et à mesure
- chaque instruction est exécutée sur la base de l'état de la mémoire laissé par l'instruction précédente.

Dans l'exemple précédent, la mémoire est initialement vide.

- l'instruction numérotée 1 est exécutée et modifie la mémoire qui contient alors l'association de la valeur `1` à `x`.
- l'instruction numérotée 2 est exécutée en considérant que la valeur associée à `x` est `1`. La mémoire est modifiée.
- la mémoire finale contient l'association de la valeur `1` à `x` et l'association de la valeur `3` à `y`.

Si l'exécution d'une instruction produit une erreur, la suite de la séquence n'est pas exécutée.

Par exemple dans l'exemple suivant la première ligne produit une erreur de nom. La mémoire reste vide et les deuxième et troisième lignes ne sont pas exécutées. La mémoire finale est vide.

```
x = x + 1
y = 1
z = 2
```

Que se passe-t-il si on utilise dans un bloc de code une expression dont la valeur n'est pas affectée à une variable ?

```
x = 1
x + 2 # et non y = x + 2
```

Dans ce cas l'expression `x+2` est évaluée à la valeur `3`. Comme la valeur `3` n'est pas ajoutée dans la mémoire en l'absence d'affectation, cette valeur est perdue. La mémoire finale contient l'association de la valeur `1` à `x` et c'est tout. Néanmoins ce n'est pas une erreur de syntaxe.

NB Les identificateurs de variables suivent les mêmes règles de syntaxe (lexicales) que les identificateurs décrits dans les supports sur les fonctions.

4 Chaînes formatées

On a parfois besoin de construire des chaînes de caractères à partir de valeurs numériques.

Par exemple, on peut avoir calculé un nombre de cartons, stocké dans une variable `nb` de type entier, et avoir besoin de calculer une chaîne de la forme `"ce jour 12 cartons"` si la valeur associée à `nb` est `12`.

La mauvaise manière de faire est de convertir la valeur de `nb` en une valeur de type `str` et d'utiliser une concaténation: ~~`"ce jour " + str(nb) + " cartons"`~~.

L'évaluation de cette expression produira bien la valeur souhaitée, mais l'expression n'est guère lisible. De plus on va souvent oublier de convertir l'entier en chaîne, ce qui produira une erreur de type.

Python propose à la place des **chaînes formatées** :

```
>>> nb = 12
>>> f'ce jour {nb} cartons'
'ce jour 12 cartons'
```

On notera :

- le caractère `f` qui précède le délimiteur de chaîne
- les accolades qui délimitent la variable dont on souhaite inclure la valeur dans la chaîne.

On peut utiliser plus généralement la valeur d'une expression :

```
>>> f'demain {nb + 10} cartons'
'demain 22 cartons'
```


5 Memento

- une variable est un nom qui désigne une valeur
- l'affectation associe une valeur à une variable
- l'état de la mémoire représente l'ensemble des associations identificateur/valeur du processus Python en cours dans l'interpréteur ou d'un bloc de code

On découvre comment utiliser les variables quand on programme des fonctions. On verra en particulier :

- deux types de variables : les variables locales et les variables globales
- la notion de portée des variables

1 Du besoin des variables quand on écrit des fonctions

Jusqu'à présent nous avons vu des fonctions dont le corps est réduit à une instruction `return`. Dans le cas général, le corps de la fonction est un bloc de code de plusieurs lignes, qui utilise des variables et des affectations (et bien d'autres choses dans les chapitres suivants).

1.1 Un exemple de fonction avec variable locale

Supposons qu'on veuille écrire une fonction qui calcule l'impédance[1] d'un condensateur en fonction de sa capacité C et d'une fréquence donnée f. La formule est la suivante :

$$x_c = \frac{1}{\omega C}$$

avec :

$$\omega = 2\pi f$$

où ω désigne la pulsation.

Tel que les formules sont présentées, il semble naturel de calculer d'abord la pulsation, puis de calculer l'impédance à partir de la pulsation.

Pour ce faire, on va utiliser une variable pour stocker la valeur de la pulsation :

```
from math import pi

def impedance_condensateur(capacite:float, frequence:float) -> float:
    """ Calcule l'impédance d'un condensateur.
    [...]
    """
    pulsation = 2 * pi * frequence    #1
    return 1 / (pulsation * capacite) #2
```

NB On notera l'utilisation de la constante `pi` du module `math` et surtout pas d'une constante approximative écrite à la main comme `3.14`.

La variable `pulsation` étant définie dans la fonction `impedance_condensateur`, on dit qu'elle est **locale** à la fonction.

Les paramètres d'une fonction sont des variables locales particulières.

Évaluer un appel à la fonction `impedance_condensateur(25e-9, 80000)` se fera de la manière suivante :

- les valeurs des paramètres sont évaluées
- la mémoire contient initialement :

capacite	25e-9
frequence	80000

- le bloc de code constitué des lignes numérotées 1 et 2 est exécuté en se basant sur cette mémoire, ce qui déclenche l'exécution de la ligne numérotée 1
- l'instruction numérotée 1 est exécutée, l'expression `2 * pi * f` est évaluée approximativement à la valeur `502654`[2]. Cette valeur est associée dans la mémoire à la variable `pulsation`

capacite	25e-9
frequence	80000
pulsation	502654

[1] Cet exemple n'a pas été écrit par une électronicienne, mes excuses s'il n'est pas correct.
[2] 502654.8245743669 précisément

- l'instruction numérotée 2 est exécutée, l'expression `1 / (pulsation * capacite)` est évaluée approximativement à la valeur `79.6`, valeur renvoyée par la fonction.
- l'appel termine et la mémoire associée à l'appel de fonction disparaît.

Notons que, sans variable locale, nous aurions écrit :

```
return 1/(2 * pi * frequence * capacite)
```

ce qui produirait le même résultat mais se lit moins bien.

Une propriété des variables locales est qu'elles sont visibles uniquement par la fonction dont le corps les contient.

Dans l'exemple précédent, il est inutile d'essayer d'accéder à la valeur de `pulsation` dans la console, ou depuis une autre fonction.

1.2 Un exemple de fonctions avec variable globale

Supposons qu'on veuille écrire 2 fonctions, qui calculent respectivement la force gravitationnelle F et la valeur de la gravitation g définies par les formules suivantes :

$$F = \frac{G \times m_A \times m_B}{d^2} \qquad g = \frac{G \times m}{R}$$

dans lesquelles $G = 6.67 * 10{-}11SI$, d désigne une distance entre planètes, m, m_1 et m_2 des masses de planètes et R le rayon d'une planète.

La fonction `gravitation` prend en paramètre la masse de la planète `masse` et son rayon `rayon`. Mais que faire de la constante G ?

Une première mauvaise solution consiste à écrire : ~~`return (6.67e-11 * masse) / rayon`~~

En effet on ne comprend pas en lisant le code quel est le sens de la valeur `6.67e-11`. Une bonne pratique consiste à nommer les constantes par un nom porteur de sens (`pi` dans l'exemple précédent).

Une deuxième mauvaise solution consiste à utiliser une variable locale :

~~`G = 6.67e-11`~~

~~`return (G * masse) / rayon`~~

En effet, la fonction `force_gravitationnelle` a aussi besoin de la valeur de `G` pour écrire son calcul. Et il est préférable qu'elle utilise exactement la même valeur, qu'il est logique de partager entre les 2 fonctions.

Une dernière mauvaise solution consiste à utiliser un paramètre pour G : un paramètre sert à faire varier les valeurs, et ne convient pas pour une constante.

La solution consiste alors à utiliser une **variable globale** au fichier qui contient les deux fonctions `force_gravitationnelle` et `gravite`. Dans ce cas la valeur de `G` sera accessible depuis n'importe quelle fonction dans le fichier.

```
G = 6.67e-11

def gravite(masse:float, rayon:float) -> float:
    """
    [...]
    """
    return (G * masse) / rayon
def force_gravitationnelle(distance:float, masse_A:float, masse_B:float) -> float:
    """
    [...]
    """
    return  (G * masse_A * masse_B) / (distance**2)
```

Les variables globales apparaissent dans une mémoire globale au module. En première semaine nous n'avons traité que des exemples pour lesquels cette mémoire globale était vide.

Lors de l'appel d'une fonction, la mémoire contient la valeur des paramètres **et** la valeur des variables globales. Un exemple d'appel de fonction avec variable globale est décrit dans la section suivante.

2 Variables globales, variables locales

Une variable est dite *globale* si elle est définie directement dans le fichier (par exemple un fichier qui contient une ou plusieurs fonctions).

Si par exemple le contenu du fichier `mon_fichier.py` est le suivant :

```
C = 2
```

alors `C` est une variable globale.

Une variable est dite *locale* si elle est définie dans une fonction.

Si par exemple on enrichit le contenu du fichier `mon_fichier.py` de la manière suivante :

```
C = 2
def f(x : int) -> int:
    y = x + 3
    return y * 2
```

alors `y` est une variable locale à la fonction `f`.

Les paramètres d'une fonction sont aussi des variables locales à cette fonction.

3 Portée des variables

La *portée* d'une variable est la partie du code dans laquelle on peut l'utiliser (son nom est connu, on peut accéder à sa valeur).

Pour une variable *globale*, la portée est *l'ensemble du fichier*. La variable peut être utilisée y compris dans le corps des fonctions du fichier.

Pour une variable *locale à une fonction*, la portée est uniquement *la fonction*.

Modifions un peu le contenu du fichier précédent en remplaçant dans le corps de la fonction `f` le littéral `2` par la variable `C` :

```
C = 2
def f(x : int) -> int:
    y = x + 3
    return y * C
```

`C` est une variable globale, sa valeur peut donc être utilisée dans le corps de `f`.

Par contre `y` est locale à `f`, tenter d'utiliser sa valeur en dehors de `f` produira une erreur.

Le code entré dans la console est soumis au même mécanisme de portée :

```
>>> C = 2
>>> def f(x : int) -> int:
...     y = x + 3
...     return y * C
...
>>> f(4)
14
>>> y
[...]
NameError: name 'y' is not defined
>>> C
2
```


Déroulons l'appel à la fonction `f(4)` en visualisant l'état de la mémoire.

Avant l'appel à `f` la mémoire contient uniquement les variables globales :

C	2

Juste après l'évaluation des arguments de `f`, la mémoire contient les variables globales et les paramètres de `f` :

x	4
C	2

Pendant l'exécution du corps de `f` la mémoire contient aussi les variables locales :

x	3
C	2
y	7

Après la fin de l'appel la mémoire ne contient à nouveau plus que les variables globales.

4 Et si une variable locale porte le même nom qu'une variable globale?

Supposons que par inadvertance, on modifie le code de `f` en redéfinissant la valeur de `C` dans la fonction :

```
C = 2
def f(x : int) -> int:
    C = 3
    y = x + C
    return y * 2
```

La variable `C` apparaît à la fois comme variable globale au fichier et comme variable locale à `f`.

Dans ce cas, la règle dans les langages de programmation est que durant le temps de l'exécution de `f`, *les variables locales masquent les variables globales*.

Lors de l'appel à `f(4)`, l'exécution commence avec la mémoire contenant les variables globales :

C	2

Juste après l'évaluation des arguments de `f`, la mémoire contient les variables globales et les paramètres de `f` :

x	4
C	2

L'exécution de `C = 3` met à jour la version locale de la variable `C`.

x	4
C	3

L'exécution de `y = x + C` utilise la version locale de `C`.

x	4

C	3
y	7

L'appel `f(4)` renvoie bien la valeur `14`.

Une fois l'appel terminé, la variable globale `C` a toujours la valeur d'avant appel :

```
>>> C = 2
>>> def f(x : int) -> int:
...     C = 3
...     y = x + C
...     return y * 2
...
>>> f(4)
14
>>> C
2
```

Il est très facile de se mélanger les pinceaux quand on utilise le même nom pour des variables globales et des variables locales. La règle dans l'UE (*cf.* plus bas) est de ne pas le faire.

5 Et si deux fonctions utilisent le même nom de variable locale ?

On pourra écrire par exemple :

```
C = 2
def f(x : int) -> int:
    y = x + 3
    return y * C
def g(z : int) -> int:
    y = 3 * z
    return y ** 2
```

La variable `y` locale à `f` est visible uniquement par `f`, la variable `y` locale à `g` est visible uniquement par `g`.

Lors de l'évaluation de `f(4) + g(2)` :

- la variable `y` locale à `f` prendra la valeur `7` lors de l'appel à `f(4)`
- la variable `y` locale à `g` prendra la valeur `6` lors de l'appel à `g(2)`

Cet usage des variables locales est courant et ne pose pas de problème.

6 Et si un paramètre porte le même nom qu'une variable globale ?

Supposons que par inadvertance, on modifie la signature de la fonction `f` en passant la valeur de `C` en paramètre :

```
C = 2
def f(x : int, C : int) -> int:
    y = x + 3
    return y * C
```

La variable `C` apparaît à la fois comme variable globale au fichier et comme paramètre de `f`.

Comme un paramètre est une variable locale, la règle de masquage s'applique.

Lors de l'appel à `f(2 * C, 5)`, l'exécution de `f` commence avec la mémoire contenant les variables globales :

C	2

Université de Lille
Variables et portée, fonctions avec variables
2025-2026
UE PROG, L1S1 MI

L'évaluation de l'argument `2 * C` utilise donc la valeur de la variable globale `C` : `x` prend la valeur `4`.

Juste après l'évaluation des arguments de `f`, la mémoire contient les variables globales et les paramètres de `f`, la valeur locale de `C` venant masquer la valeur globale :

x	4
C	5

L'exécution de `y * C` utilise la valeur locale de `C`, soit `5`. L'appel `f(2 * C, 5)` renvoie donc la valeur `35` (évaluation de `7 * 5`). Une fois l'appel terminé, la variable globale `C` a toujours la valeur d'avant appel : `2`.

Il est très facile de se mélanger les pinceaux quand on utilise le même nom pour une variable globale et un paramètre. La règle dans l'UE (*cf.* plus bas) est de ne pas le faire.

7 Et si deux fonctions utilisent le même nom de paramètre ?

On pourra écrire par exemple :

```
C = 2
def f(x : int) -> int:
    y = x + 3
    return y * C
def g(x : int) -> int:
    y = 3 * x
    return y ** 2
```

De même que pour des variables locales de même nom, cet usage ne pose pas de problème.

8 Bonnes pratiques de programmation

Il convient de toujours choisir des noms porteurs de sens pour les variables, locales comme globales.

8.1 Variables globales et constantes

Dans cette UE, il est d'une manière générale *interdit* d'utiliser des variables globales dans les exercices de programmation qui impliquent des fonctions (hors des exercices de manipulation).

Les seules variables globales tolérées sont des variables dont le but est de représenter une *constante*.

Une telle variable :

- a un nom écrit en majuscules (convention Python pour les variables globales)
- est affectée une et une seule fois en début de module.

Par exemple :

```
NB_SECONDES = 60
NB_MOIS = 12
```

On n'utilisera ces noms de "constantes" :

- ni comme des noms de paramètres de fonctions
- ni comme des noms de variables locales

8.2 Paramètres

On peut être tenté d'utiliser un paramètre comme une variable locale, au lieu de définir un nouveau nom :

Département Informatique de la Faculté des Sciences et Technologies

```
C = 2
def f(x : int) -> int:
    x = x + 3
    return x * C
```

Dans l'exemple précédent on a écrit `x = x + 3` et `return x * C` au lieu de `y = x + 3` et `return y * C`. On a donc modifié la valeur du paramètre au lieu de définir une nouvelle variable locale.

Vous verrons en fin d'UE qu'il est parfois souhaitable d'écrire des fonctions qui modifient la valeur de leurs paramètres. Mais pour débuter il est recommandé de ne pas affecter ainsi les paramètres des fonctions dans leur corps.

1 Les fonctions intermédiaires

Une bonne pratique en programmation consiste à décomposer le problème initial en sous-problèmes, chacun étant codé par une fonction.

Les fonctions qui codent un sous-problème sont appelées **fonctions intermédiaires**.

1.1 Exemple de fonction intermédiaire

On reprend l'exemple du calcul d'impédance du support précédent.

Supposons qu'on souhaite cette fois calculer l'impédance d'une bobine[1] en fonction de son inductance L et d'une fréquence donnée f. La formule est la suivante :

$$x_b = \omega L$$

avec :

$$\omega = 2\pi f$$

où ω désigne la pulsation.

On retrouve la pulsation déjà calculée pour le calcul de l'impédance d'un condensateur.

Ce serait une mauvaise pratique de calculer la valeur de la pulsation deux fois, une fois dans la fonction `impedance_condensateur` et une fois dans la fonction `impedance_bobine`.

En effet dupliquer du code peut signifier dupliquer des erreurs. De plus faire évoluer du code dupliqué n fois donne n fois plus de travail.

On va donc utiliser une fonction intermédiaire pour calculer la pulsation.

```
from math import pi

def pulsation(frequence:float) -> float:
    [...]
    return 2 * pi * frequence

def impedance_condensateur(capacite:float, frequence:float) -> float:
    """ Calcule l'impédance d'un condensateur.
    [...]
    """
    pulsat = pulsation(frequence)
    return 1 / (pulsat * capacite)

def impedance_bobine(inductance:float, frequence:float) -> float:
    """ Calcule l'impédance d'une bobine.
    [...]
    """
    pulsat = pulsation(frequence)
    return pulsat * inductance
```

Dans le fichier final on a 2 fonctions principales, et une fonction intermédiaire.

On notera que les fonctions principales réalisent chacune un appel à la fonction intermédiaire. La première ligne du corps des fonctions est :

```
    pulsat = pulsation(frequence)
```

Cet appel peut perturber, car jusqu'à présent nous n'avons vu que des appels de fonction qui passaient des valeurs littérales aux paramètres, par exemple `pulsation(80000)`.

Comment cet appel `pulsation(frequence)` est-il évalué ?

[1] excuses renouvelées aux électroniciens

Lors d'un appel, le contenu de la mémoire de la **fonction dite appelante** (par exemple `impedance_condensateur`) est transmis à la **fonction dite appelée** (`pulsation`).

Si on reprend l'appel du support précédent : `impedance_condensateur(25e-9, 80000)` :

- en début d'appel la mémoire contient initialement :

`capacite`	`25e-9`
`frequence`	`80000`

- l'instruction `pulsat = pulsation(frequence)` est exécutée
  - la partie droite de l'affectation est évaluée, l'expression `pulsation(frequence)` est évaluée
  - l'expression `frequence` passée au paramètre dans l'appel est évaluée par rapport à l'état de la mémoire : on y trouve que `frequence` est associé à la valeur `80000`
  - l'appel `pulsation(80000)` est évalué approximativement à la valeur `502654`
  - la mémoire contient en fin d'exécution de l'affectation :

`capacite`	`25e-9`
`frequence`	`80000`
`pulsat`	`502654`

- la suite se déroule comme précédemment.

1.2 Quand utiliser une fonction intermédiaire ?

On peut avoir besoin d'une fonction intermédiaire pour plusieurs raisons :

Le problème est gros : une fonction qui le coderait intégralement contiendrait de nombreuses lignes de code et serait par là incompréhensible. On estime souvent que le corps d'une fonction (docstring exclue) ne doit pas dépasser 10 lignes.

Le problème est compliqué : on ressent le besoin de tester certains calculs intermédiaires, et donc d'isoler ces calculs dans une fonction avec tests.

On est perdu : on n'arrive pas à trouver la source d'une erreur. Quand on a du mal à tester et déboguer une fonction, c'est souvent qu'elle est trop longue, et mieux vaut la décomposer. On trouvera plus facilement l'erreur dans la fonction intermédiaire dont les tests ne passent pas. L'erreur n'est pas a priori dans les fonctions intermédiaires dont les tests passent.

Le problème contient des calculs redondants : c'est une très mauvaise pratique de dupliquer des morceaux de code. En effet, si on veut faire évoluer ce code ou corriger une erreur qu'il contient, il faudra faire la modification partout où ce code apparaît. C'est fastidieux et on risque de rater des modifications à faire.

D'une manière générale, une fonction permet d'encapsuler un calcul et de le tester. C'est la brique abstraite de base, il ne faut pas s'en priver.

2 Méthode pour écrire une fonction

Cette méthode essaie de préciser la manière générale d'aborder un énoncé.

1) **lecture de l'énoncé**

- lire l'énoncé attentivement, repérer les éléments clés
- quelles sont les données variables du problème (+ quels types ?), quelles sont les constantes ?
- que doit-on calculer ? (quel type ?)
- quels sont les calculs appliqués aux données du problème pour le résoudre ? Quels sont les comportements de la fonction ?
- dispose-t-on d'opérateurs ou de fonctions prédéfinies pour faire des calculs ?
- poser des questions si l'énoncé est flou (sous-spécification)
- à l'issue de cette étape on doit pouvoir prendre au moins un exemple : choisir des valeurs littérales pour les données du problème et savoir comment calculer une valeur littérale, réponse au problème

2) **écriture de la signature de la fonction** avec ses annotations de type

Université de Lille
Écrire des fonctions
2025-2026
UE PROG, L1S1 MI

- chaque donnée du problème devient un paramètre
- les constantes ne sont pas des paramètres mais des variables globales

3) **écriture de la documentation**

- sans en faire un roman
- en réfléchissant bien à la précondition
- en prévoyant des tests (les exemples ont déjà été réfléchis)

4) **écriture du code de la fonction**

- identifier les fonctions intermédiaires nécessaires, les coder et les tester en appliquant à nouveau cette méthode.
- coder la fonction

5) **exécuter les tests**

6) **si tous verts** : fini ou réusinage + retour à 5)

7) **si rouge** :

- revenir à 3) si l'erreur vient des tests
- revenir à 4) si l'erreur vient du code
- utiliser le débogueur si on ne trouve pas l'erreur

Autre possibilité pour 4) : commencer sans fonctions intermédiaires et réusiner (= récrire le code à comportement identique) le code est modifié dans un second temps pour introduire les fonctions intermédiaires.

Département Informatique de la Faculté des Sciences et Technologies